Honours Project Proposal
Anti Compiler: an Educational Tool in First Year Programming

Kymberly Fergusson,
Monash University, Clayton, Australia
email: kef@csse.monash.edu.au
Computer Science & Software Engineering

May 16, 2002

Contents

List of Tables

Chapter 1

Introduction

Semantic errors in programming are extremely difficult to locate and solve, unlike syntax errors which are located by the compiler. Semantic errors often generate no errors at compile or run time, depending on language, but cause the program to behave in an unexpected way. Students learning to program often have great difficulty analysing their programs for semantic errors, getting bogged down in the language syntax. Many of the sematic errors students, and experienced programmers make are extremely common, an infinite loop caused by a failure to increment the loop variable is just one example. It would be useful to automatically process code to identify these common errors, for both the students/programmers writing the programs, and the teacher who could use the common errors as a focus for refining teaching methods. Running a program to highlight potential problems could simplify error-detection and correction.

This project aims to discover if it would be feasible for a conversion program translating a more complex language (C) to a more familiar language, to automatically detect common semantic errors. Based on the findings, a software tool may be developed that would analyse and diagnose possible semantic errors in programs written by students learning programming. The 'anti-compiler' would take code written in the C programming language and convert it into an English or 'pseudo code' algorithm which would allow the students to see the underlying algorithm better than when written in a syntax they are just beginning to learn.

The C programming language has notoriously terse and obscure syntax, meaning that students beginning to learn the language have a lot of trouble finding and solving problems. Converting a program into a more descriptive language may help students to identify and fix their semantic errors quickly. It may also help them develop a better understanding of programming concepts and debugging procedures, as well as solve immediate problems in their code. Learning to step through code, and being able to trace the flow of a program

is essential in debugging. Following the pseudo-code of their program, rather than getting 'bogged down' in syntax, should help develop this skill.

To develop the anti-compiler, it is necessary to analyse examples of student programs to determine how similar their algorithms are. From this, common semantic problems may be identified, enabling us to target problems in the underlying understanding of algorithms, and representing these algorithms in the C programming language. It is also necessary to design a good pseudo-language in order for the more verbose representation of the students' programs to be understood quickly and clearly, minimising misunderstandings. It is known that more verbose languages lead to a higher ambiguity, as there are many more possible ways of interpreting meaning, thus careful design is necessary to minimised misunderstandings.

From this analysis and language design, it should be possible to determine whether it is feasible for an automatic tool converting C programs to a pseudo-language to reliably identify semantic errors. A tool may then be designed and written to parse student code and generate the pseudocode representation, checking for common semantic problems (as found in the analysis of the student code), and highlighting possible errors.

Chapter 2

Research Context

Much research has found that the development of languages is specifically driven by technological advances, by experts in the field, with no or little regard for beginner programmers [MC96, Pan00, Mur93]. This tends to result in very specific and complex syntaxes and representations of programming constructs. This causes programmers to present solutions unnaturally [Pan00].

Languages designed with Human Computer Interaction (HCI) principles in mind should be more accessible to beginners. In particular the most important principle is: "to speak the user's natural language" [Pan00]. Many studies have shown that novice programmers resort to natural language solutions when they reach an impasse [BS85, AE99].

It has been reported in many studies that the more like the natural language of the user the programming language is, the easier it is to learn [PRM01, Mur93]. The closer the language is to the novice's natural problem solving language, the easier the novice can find problems, design solutions, which would result in fewer overall bugs in the code [MC99, McI00].

Perkins et. al.[PHH+89], categorised novice learners into "Stoppers" and "Movers", those who stop when at an impasse and do not try to look for a solution, and those who continually try something different, often repeating solutions. It was found in this study that with appropriate prompting, students of both types would often continue and find a solution.

A complex language such as C is very difficult as both a language to learn and to teach to novice programmers, as they need to deal with a terse non-natural syntax, at the same time learning programming constructs. Development of solutions to programming problems is often encouraged to include designing the solution in a pseudo-language first. It may be useful for students to be able to compare their design, with a more natural representation of what their

C programs are actually doing. It may also provide the encouragement that 'Stoppers' need to get past an impasse.

Classification of types of errors is important in understanding how novice programmers learn programming, and common misconceptions [SSP85]. Several studies have classified types of errors in different ways, the most common just splitting errors into semantic or logic errors vs syntax errors [McI00]. Spohrer, Soloway and Pope classified a sub-type of semantic errors as 'goal drop out' and 'goal fragmentation' [SSP85]. These two categories occur where students 'merge' goals together, resulting in a more complex design when using loops and control structures where multiple sentinels need to be updated or tested.

One other classification of semantic errors by Pea [Pea86], identifies 'parallelism' where conditions of a loop are continually tested, instead of once per iteration, 'intentionality' where the computer is attributed to have some intelligence to look ahead in the code and evaluate such, and finally, 'egocentrism' where students acted (not necessarily believed) that the computer had intelligence to 'know' what they meant without their intentions being specified in the code. This behaviour was also noted by Perkins et. al. [PHH+89].

Many studies have been conducted into the more complex programming constructs such as loops and Boolean conditionals, and have found that there are ambiguities when translating the user's natural language knowledge of keywords such as `WHILE` and the Boolean operators `NOT` and `AND` in particular, to the programming language. It was commonly found that conditions on While loops were expected to be evaluated at all times, dropping the flow of control out of the loops as soon as the condition is false [MC99, SSP85, Pea86].

In the analysis of sample programs, it will be necessary to classify semantic errors in some way, to facilitate the design of a program which can identify these errors based on type. The representation of looping and Boolean constructs will require detailed consideration, as these are the two most common areas of semantic errors.

One other interesting proposal, put forward by Spohrer et al [SSP85], is that in 200 separate solutions to a problem by novice programmers, if you received two the same, they would be from students who collaborated in designing the solution. It will be educational to see if students indeed create very different programs solving the same problem, as it may complicate the analysis and detection of semantic errors by an automatic program.

Chapter 3

Method and Plan

3.1 Analysis of student programs

Discovering the similarity of students' algorithmic development, is necessary to figure out if a semantic analysis program is feasible. Analysing beginner programmers' code will highlight which semantic problems are most common, and help with the design of an automated tool which could identify and highlight the problem.

Code needs to be collected at various stages of development by first year students studying CSE1301, but in order to be analysed for semantic errors, it must be syntactically correct, as syntax errors will not be dealt with - the compiler already checks, analyses and reports errors for this. Collecting examples at varying stages will allow identification of patterns in development, and may enable identification of where in the implementation of algorithms, semantic errors occur, if there is a commonality across the program segments collected.

3.2 Pseudo-language design

Reviewing literature on pseudo-language design is crucial, as much research has been done into various representations of programming constructs, and common semantic errors. This will aid in the design of a verbose, clear and unambiguous language to which the C program will be converted. The pseudo-language will be developed from existing research on language design and semantic representation of programming constructs for beginners.

The language may be tested on beginners to see how clear the representations of the more difficult programming constructs are in the new pseudo-language.

3.3 Anti-Compiler

Once the pseudo-language has been designed, a parser will need to be written to parse the C programs, and a conversion tool to convert them to the pseudo-language. When developed, this tool will need to be tested on the original programs, and on common algorithms with introduced errors to make sure the conversion works well.

Further development of the program if feasible, will include semantic error checking, to automatically detect and highlight the semantic errors in a C program. This will also need to be tested on the original sample collected, as well as on common algorithms, with and without introduced errors, to ensure that the detection is reliable.

The parser and conversion tool will be custom written in Perl as a module, as this is an excellent language for parsing text. Matching common semantic problems may also use various existing Perl modules for pattern matching.

The program will be tested on the original programs collected for analysis, and on further collected programs from beginner programmers, increasing in complexity. These programs will all be written in C, and are required to be syntactically correct.

3.4 Proposed Chapters for Thesis

1. Introduction
   (a) Brief background
   (b) Purpose of research
2. Beginner programming
   (a) Psychology of learning programming
   (b) Basic pseudo-language issues
   (c) Common semantic errors
   (d) Analysis of beginner programming habits
3. Anti-compiler
   (a) Pseudo-language design
   (b) Parsing
   (c) Representing a C program in pseudo-language
   (d) Testing
   (e) Results
4. Conclusion and Future Work
5. Bibliography
6. Appendices
   (a) Sample data
   (b) Pseudo-language
   (c) Programs

3.5 Proposed Time Line

Semester One

2	14.03.2002	Ethics Proposal #1
5	08.04.2002	Research Proposal outline completed
7	22.04.2002	Literature overview for proposal
8	02.05.2002	Research Proposal
10	13.05.2002	Collect student programs
11	20.05.2002	Pseudo-language researched and begun design
12	27.05.2002	Literature Review outline completed
12	30.05.2002	Interim Presentation draft complete
13	03.06.2002	Analysis of student programs mostly complete
13	06.06.2002	Interim Presentation
14	10.06.2002	Analysis of student programs complete
14	13.06.2002	Literature Review Draft
15	17.06.2002	Thesis chapter 2 - beginner programming
15	20.06.2002	Pseudo-language complete
16	20.06.2002	Begin coding

Semester Two

1	25.07.2002	Finalise Literature Review
3	01.08.2002	Literature Review
4	12.08.2002	Begin testing anti-compiler
5	15.08.2002	Thesis chapter 1 - introduction
6	22.08.2002	Thesis chapter 4 - anti-compiler
7	29.08.2002	Thesis appendices
8	12.09.2002	First Thesis Draft
11	07.10.2002	Finish coding and testing
13	21.10.2002	Prepare final presentation
13	25.10.2002	Finalise Thesis and logbook
14	28.10.2002	Final Presentation
14	01.11.2002	Research Log Book
15	05.11.2002	Final Thesis
15	06.11.2002	Finalise Web site
16	11.11.2002	Web site

Table 3.1: Proposed Project Time Line

Chapter 4

Relevance

As seen in Chapter 3, much research has been conducted trying to discover how to design languages so that novices can learn more naturally. This research only related to new language design. So far, not much research has been found analysing methods or the feasibility of automated tools to make existing languages more accessible.

The analysis of students' code will prove enlightening, allowing us to see if there is any similarity in developed algorithms/solutions to a problem, or if as Sporher et.al. [SSP85] claim, the only similar code in 200 submissions is where students have collaborated. If they are indeed as varied as reported, it may pose a difficulty in reliably detecting semantic errors.

In the analysis of the submissions, errors will have to be classified, which will make designing the tool easier and more focused on discovering the common types of semantic errors made.

There has been research done on how to teach existing complex languages in a way that is easier for the students to comprehend, but an automated tool that can represent a student's solution in a language that they understand more would be enormously helpful in their learning process. It may provide that encouragement and knowledge that can get them past an impasse.

A tool such as the one proposed, would also be helpful for instructors wanting to improve the focus of their teaching by providing them with information on common misconceptions/errors made by students.

The pseudo-language will need to be carefully designed, so as to be accessible to students, utilising their natural language knowledge, but minimising ambiguities between it and the programmer's natural language. Careful analysis of looping and conditional constructs is needed, to decide on the clearest way to

translate these from an imperative language (C), to the pseudo language.

Combining the analysis of novice programmer solutions, the design of an intuitive and natural pseudo representation of the complex language of C, and an automated tool will have many benefits, both to the programmers and the instructors.

Bibliography

[AE99] Bruckman A. and E. Edwards. Should we leverage natural-language knowledge? An analysis of user errors in a natural-language-style programming language. In *CHI'99 Papers*, 15-20 May 1999.

[BS85] J. Bonar and E. Soloway. Preprogramming knowledge: A major source of misconceptions in novice programmers. *Human-Computer Interaction*, 1(2):133–161, 1985.

[MC96] L. McIver and D. Conway. Seven deadly sins of introductory programming language design. In *Software Engineering: Education and Practice*. IEEE Computer Society Press, 1996.

[MC99] L. McIver and D. Conway. Grail: A zero'th programming language. In G Cummings, T. Okamoto, and L. Gomex, editors, *7th International Conference on Computers in Education ICCE'99*, volume 2, pages 43–50. IOS Press, November 1999.

[McI00] L. McIver. The effects of programming language on error rates of novice programmers. In *12th Annual Meeting of the Psychology of Programming Interest Group*, April 2000.

[Mur93] J. Murnane. The psychology of computer languages for introductory programming courses. *New Ideas in Psychology*, 11(2):213–228, 1993.

[Pan00] B. Pane, J. Myers. The influence of the psychology of programming on a language design: Project status report. 12th Annual Meeting of the Psychology of Programming Interest Group, 10-13 April 2000. PPIG 2000.

[Pea86] R. D. Pea. Language-independent conceptual "bugs" in novice programming. *Educational Computing Research*, 2(1):25–36, 1986.

[PHH+89] D.N. Perkins, C. Hancock, R. Hobbs, F. Martin, and R Simmons. *Conditions of Learning in Novice Programmers*. Lawrence Erlbaum Associates, 1989.

[PRM01] J. F. Pane, C. Ratanamahatana, and B. A. Myers. Studying the language and structure in non-programmers' solutions to programming problems. *International Journal of Human-Computer Studies*, 54:237–264, 2001.

[SSP85] J. C. Spohrer, E. Soloway, and E. Pope. Where the bugs are. In *CHI'85 Conference on Human Factors in Computing Systems*, pages 261–279, April 1985.